ACTION RECOGNITION USING RECURRENT NETWORKS

PROJECT WORK OF COMPUTER VISION AND IMAGE PROCESSING M – PROFESSOR LUIGI DI STEFANO

UNIVERSITY OF BOLOGNA

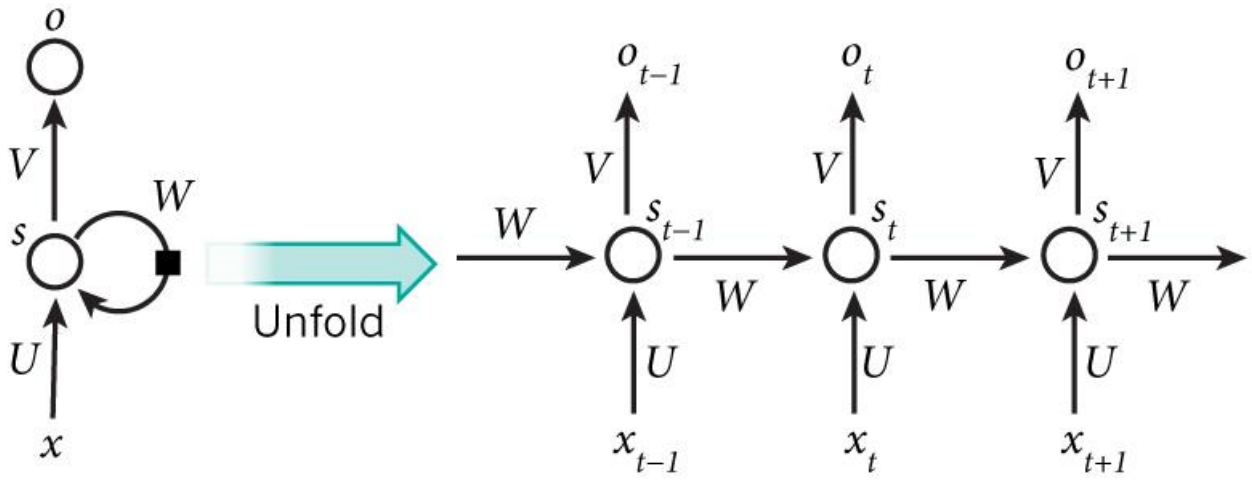
Albertazzi Riccardo
Berlati Alessandro

ACTION RECOGNITION

- In computer vision, action recognition refers to the act of classifying an action that is present in a given video
- Good test environment for RNNs, since we can consider each frame of a video as sample in the time domain.
- Samples will be processed by a CNN in order to have a latent representation of the image that we can feed to the RNN.

INTRODUCTION TO RNN

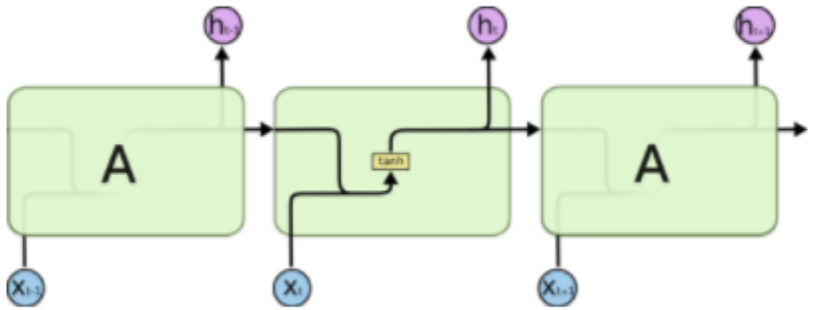
- Recurrent neural networks, or RNNs, are a family of neural networks for processing sequential data.
- The hidden state is the memory of the network and it is calculated based on the current input and the previous state.
- The output is computed based on the current state.



INTRODUCTION TO RNN – VANILLA RNN

In a Vanilla Recurrent Network the repeating module has a very simple structure, such as a single *tanh* layer. The current state in then computed as a function of the previous state h_{t-1} and the current input x_t.

$$h_t = tanh(W_t * h_{t-1} + W_x * x_t)$$

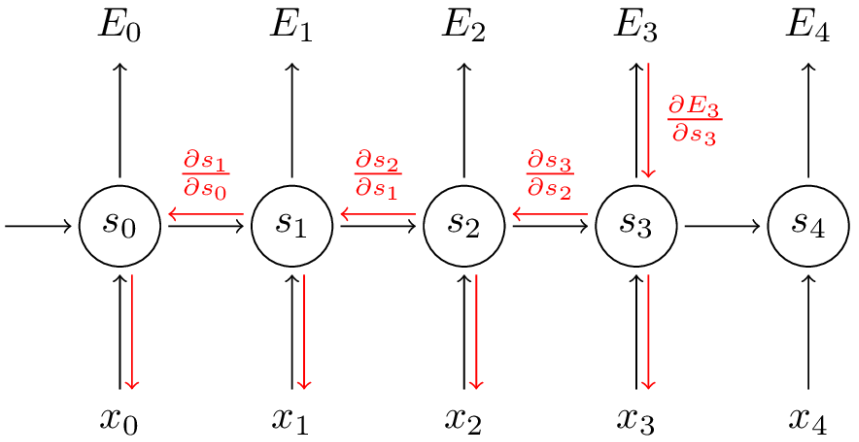


Once obtained the current state is we can compute the output state as

$$y_t = W_y * h_t$$

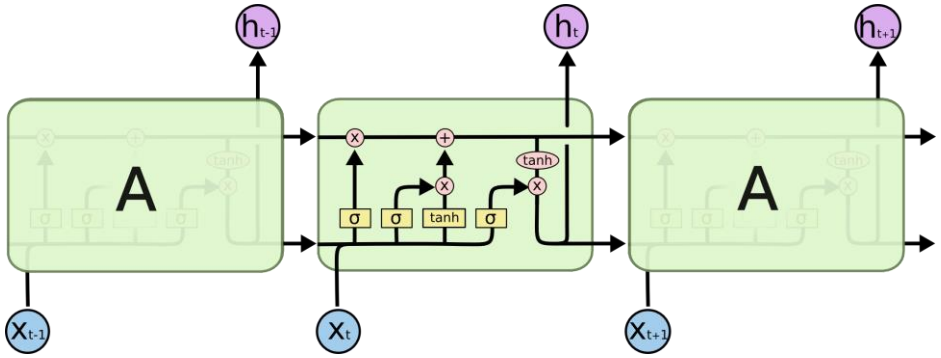
INTRODUCTION TO RNN – GRADIENT PROBLEM

- Backpropagation through time: using the chain rule of differentiation, we must backpropagate through all the timesteps that to reach the end of the graph
- Simple (Vanilla) RNNs have difficulties learning long-term dependencies due to gradient problems:
  - Vanishing gradient: gradients are multiplied by values smaller than 1; the gradient shrinks exponentially.
  - Exploding gradient: gradients are multiplied by values greater than 1, the gradient grows exponentially. Could be dealt with norm clipping



INTRODUCTION TO RNN - LSTM

- Long Short Term Memory networks are a particular type of RNN, capable of learning long-term dependencies. They are explicitly designed to avoid the long-term dependency problem.
- LSTMs have the same chain structure as Vanilla RNNs but the repeating module has a more complex structure: the key to LSTMs is the **cell state.** The cell state is one of the two hidden states that are passed through time. The LSTM has the possibility to remove or add information to the cell state, regulated by structures called gates. Gates are usually composed of a sigmoid neural net layer and a pointwise multiplication operation.



- LSTMs solve the vanishing gradient problem by eliminating the activation function for the cell state. The activation function is the identity which leads to a gradient of 1.

MACHINE SETUP

The architecture that we used:

- Hardware: virtual machine on Google Cloud
  - 8 vCPU
  - 52 GB RAM
  - 500 GB SSD
  - Nvidia Tesla P100 (16 GB VRAM)
- OS: Ubuntu 16.04 LTS
- Software:
  - CUDA 9.0 with CuDNN
  - Python 3.5.2
  - TensorFlow 1.5
  - Keras 2.1.4

OUR DATASETS

- UCF11
- UCF50
- UCF101
- HMDB51

- Raw videos that mostly involve human actions (sports, playing music, ...)
- The resolution is usually 320x240 and the length of the videos changes significantly from video to video
  - On average 200 frames

PRE-PROCESSING OF THE DATASETS

- The pre-processing step extracts a **fixed** number of frames from each video and saves the frames as JPEG files (the output filename contains an incremental index, allowing to order the images during the loading phase).
  - The number of extracted frames is fixed because a RNN requires sequences with fixed length.
- It is also possible to specify a different output format, that involves feeding a neural network (such as InceptionV3 or Yolo V2, see the next slides) the frames and saving the output as a npy file.
- The extrapolated frames are not adjacent, but they are sampled at a fixed step that varies with the length of the video (the longer the video, the higher the sampling step)

USE A CNN TO CLASSIFY EACH FRAME INDIVIDUALLY

TRAIN A CNN FROM SCRATCH

- In our first approach we didn't use recurrent networks; instead, we tried to classify each frame separately. In order to classify an entire video, we computed the mean prediction on all the video frames.

- In our first experiment we trained a convolutional neural network with randomly initialized weights. We tried different configurations of the network but we were never able to let the network learn anything.

USE A PRE-TRAINED MODEL: INCEPTIONV3

- The model, available on Keras, has been trained on the ImageNet dataset and has 0.78 top-1 accuracy.
- This is a very *deep* network: 159 layers and 24M weights!
  - This is why we decided to save the output of this model in our pre-processing phase, so that we didn't have to feed the model with our frames when training.
- We took the output of the last convolutional layer
  - To save storage space and training time, we also decided to apply a global average pooling to the convolutional output: the output of the InceptionV3 network, that becomes the input for our custom networks, is therefore a 1D vector of size 2048.

USE A PRE-TRAINED MODEL: INCEPTIONV3

- With the Inception pre-processing, our custom network becomes very simple: a fully connected layer (ranging from 256 to 512 weights) and a second fully connected layer that outputs the predictions, with some dropout in between.
- Since our network outputs one result for each frame, we tried different methods to compute the class of a video given its individual frames predictions:
  - Compute the mean of the predictions, and then taking the argmax
  - Count the top-1 predictions on each frame and take the most frequent
  - Similar results, with the first being sometimes better

Dataset	Top-1 Accuracy	Top-3 Accuracy
UCF11	88.9	98.7
UCF50	82.1	92.6
UCF101	79.0	90.4
HMDB51	46.2	69.3

USE A RNN TO CLASSIFY SEQUENCES OF FRAMES

USE A RNN TO CLASSIFY SEQUENCES OF FRAMES

- This second approach consists in classifying an entire video by enforcing the network to evaluate the dependencies between consecutive frames with a LSTM layer.

- Since training a network from scratch was very problematic, we spent most of our efforts into training a recurrent network using as input the 1D output vector from Inception. Therefore our network takes as input a 2D vector with dimensions (NB_FRAMES, INCEPTION_FEATURES) (e.g. (50, 2048)).
- The basic network that we used for testing contains one LSTM layer, one intermediate fully connected layer, and the last fully connected layer with the output predictions.

VARIATIONS THAT WE TESTED

- Changing the **output size of the LSTM**
- Change **learning rate** and **optimizer**
- Use a **stack of LSTM layers**
- Play with **recurrent dropout**, **regularizers** and **norm clipping**
- Decrease the learning rate over time when the test accuracy wasn't increasing for n consecutive epochs
- Use a variation of the LSTM layer that returns one output for each timestep; we then added a layer that computes the mean of the output sequence.
- Use a **bidirectional LSTM**
- Train with more frames per video, by padding with black frames (all zeros) at the beginning of a video
- Train a RNN using as input the frame predictions of the non-RNN approach

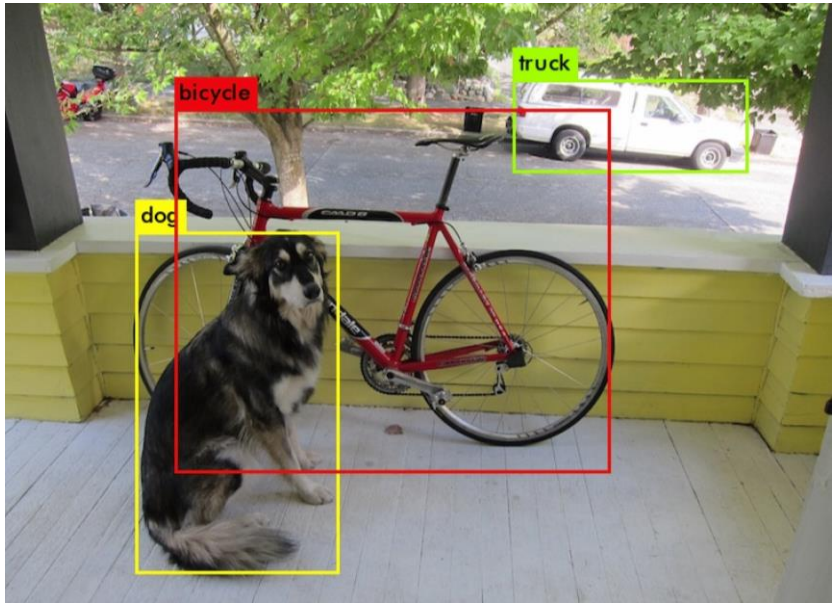
RESULTS USING RNNS

- After all these tests, we weren't able to find a recurrent network that outperforms by far the non-RNN approach.
  - We always achieved more or less the same results
  - Only on the HMDB51 we were able to achieve 48.4% accuracy using a bidirectional LSTM (+2% accuracy compared to the non-RNN approach).
- The reason why the accuracy didn't increase may be related to the input data: the Inception model does not know anything about sequences and it's therefore possible that its "answer" to similar frames of the same video is the same.
  - The RNN is then trying to learn temporal dependencies between latent vectors that "mean" the same thing.
- We also observed that with the non-RNN approach some sequences are wrongly classified and the best predicted (wrong) class has a probability near to 1. This is a confirmation that some videos are misinterpreted by Inception and therefore we cannot expect that a RNN will learn a better classification.

ATTEMPT TO USE YOLO V2 OBJECT DETECTIONS FOR TEMPORAL LEARNING

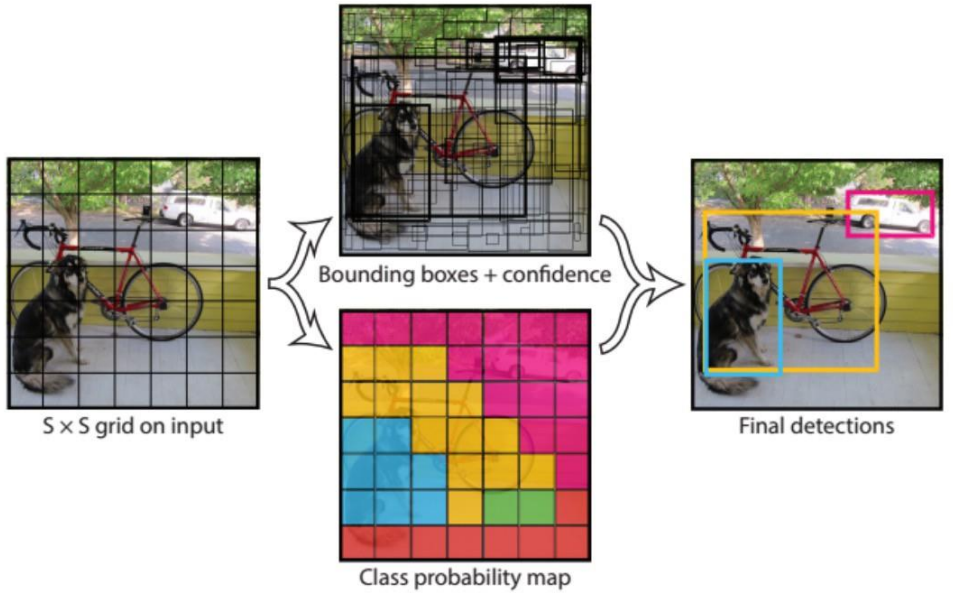
YOLO

- YOLO (You Only Look Once) is a very popular convolutional network for real time object detection. Since its output for sure will be different for different frames of a video (since objects/people move in the scene), we could use its output for our RNN training.



ACTION RECOGNITION USING YOLO

- We used a pre-trained model on the COCO dataset (80 classes).
- The output of Yolo, for one image, is a tensor of size 19x19x425
  - We can think of it as a grid of size 19x19 where for each cell we have the predictions about 5 bounding boxes and their relative class (425/5 = 85, 80 values for the one-hot classes, 5 values for the coordinates of the bounding box).
- Since the output of Yolo is a 3D tensor, we can consider that as an image
  - In our first approach we trained a convolutional network with a recurrent layer on the Yolo output and we were able to obtain 70% accuracy on UCF11.



ACTION RECOGNITION USING INCEPTION AND YOLO

- We also built a neural network that concatenates Yolo+CNN and Inception outputs, sends the sequence of concatenated vectors to a RNN, and predicts the class of the video.
  - In this case we were able to obtain a small improvement on UCF11 (91% top-1 accuracy).